Verification of a cryptographic circuit: SHA-1 using ACL2 *

Diana Toma Dominique Borrione
{diana.toma, dominique.borrione}@imag.fr
TIMA Laboratory, VDS Group, Grenoble, France

Abstract

Our study was motivated by a cooperative project aiming at the design and verification of a circuit for secure communications between a computer and a terminal smart card reader. A SHA-1 component is included in the circuit. SHA-1 is a cryptographic primive that produces - for any message, a 160 bits signature, called message digest. We automatically produce the ACL2 model for the VHDL RTL design, and apply a stepwise approach to prove theorems about each computation step, using intermediate digest functions. These functions are proven compliant with the functional specification of SHA-1.

1 Introduction

The SHA-1 is a standardized hash function [1], which processes a message up to 2^{64} bits, and produces a 160 bit message digest, with the following property: any alteration to the initial input message will result, with a very high probability, in a different message digest. The applications of this algorithm include fast encryption, password storage and verification, computer virus detection, etc. The study of SHA-1 is motivated by a project developed in cooperation with several industrial partners, aiming at the design and verification of a circuit for secure communications between a computer and a terminal smart card reader. Security considerations were at the heart of the project, it was thus of utmost importance to guarantee the correctness of the system components dedicated to security. For the SHA-1 component, formal methods were applied both for the validation of the functional specification[2], and for the verification of the implementation which is the subject of this paper.

The SHA-1 is an iterative algorithm, involving a large number of repetitions over an arbitrary number of 512-bit blocks. Property verification by model checking was first attempted [3], and provided some correctness statements about the internal design synchronization. But more powerful methods had to be applied to establish that, whatever the length of the input message, the digest is computed according to the standardized algorithm. It was thus decided to apply mechanized theorem proving technology. More precisely, we chose the ACL2 logic, for its high degree of automation, and reusable libraries of function definitions and theorem proofs [4]. The input model, being written in a subset of Common Lisp, is both executable and provable. Before investing human time in a proof, it is thus possible to check the model on test vectors, a common simulation activity in design verification which helps debug the formal model and gain designer's confidence in it. The article is organized as follows. Section 2 describes the SHA-1 algorithm and summarizes the specification model. Section 3 describes the VHDL design and its corresponding

*This is an adaptation for the ACL2 audience of the case study presented in the paper *Combining several paradigms for circuit validation and verfication*, to appear in CASSIS'04 proceedings.

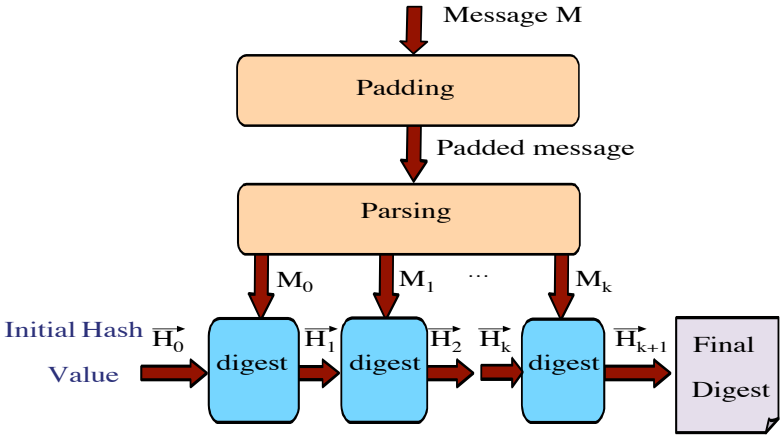

Figure 1: SHA-1 global algorithm

ACL2 model, and section 4 is an overview of the SHA-1 implementation verification vs specification model. Finally, section 5 presents our conclusions.

2 SHA-1 Algorithm

The principle of the SHA-1 is shown on Figure 1. The input message M, a bit sequence of arbitrary length $L < 2^{64}$, undergoes two preprocessing steps:

- Padding: M is concatenated by bit 1, followed by k bits 0, followed by the 64-bit binary representation of number L. k is the least non-negative solution to the equation: (L+1+k) mod 512 = 448. As a result, the padded message holds on a multiple of 512 bits.
- Parsing: The padded message is read in blocks of 512 bits. After reading each block, it must be decided if it is the last one.

The computation of the message digest is an 80-iteration algorithm over each message block, in order; a block is viewed as a sequence of 32 bit words, which are selected and combined with the contents of five 32-bit internal registers (A, B, C, D, E), using XOR and shift operations. At the start of the computation, the internal registers are initialized with predefined constants $\vec{H_0}$ $=(H_0, H_1, H_2, H_3, H_4)$. At the end of each block processing, they contain the digest obtained so far. This digest is used as an initial value for processing the next block, if there is one.

According to the SHA-1 standard [1], the digest phase operates on the 16 words W_i $(0 \leq i \leq 15)$ of a padded block in order to generate 80 words. The SHA-1 algorithm is formalized in ACL2 and the detailed model can be found in [2]. Because of hardware efficiency constraints, the VHDL design implements an alternative digest algorithm presented in the standard, which stores only sixteen W words and not eighty. We summarize below the principle of the alternative algorithm and its ACL2 formalization, knowing that we have already proven that the two methods are equivalent.

W_j $(0 \leq j \leq 79)$ and the five main variables A, B, C, D, E, are computed in the same loop, and each W_j starting from j=16 is written in place of W_j mod 16. The first sixteen words are made of the padded block itself. The 64 remaining words and A, B, C, D, E are generated as follows, where $ROTL^n$ indicates a n-bit circular left shift operation:

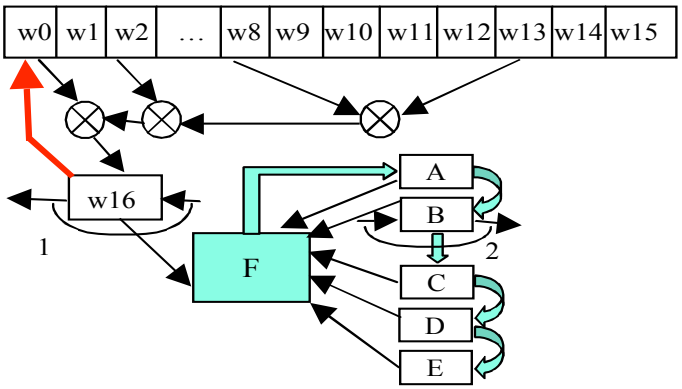

Figure 2: One computation step

for j=0 to 79 do
s=j∧MASK;
if j≥16 then
W_s=$ROTL^1$ ($W_{(s+13)\wedge MASK}$ XOR $W_{(s+8)\wedge MASK}$
XOR $W_{(s+2)\wedge MASK}$ XOR W_s);
endif;
TEMP=$ROTL^5(A)$ + F_j (B, C, D) + E + W_s + K_j;
E=D; D=C; C=$ROTL^{30}$(B); B=A; A=TEMP;
endfor.

where MASK = "00000000000000000000000000001111", F_j are functions and K_j constants defined in the SHA-1 standard. Figure 2 displays the computation for j=16.

Here is the corresponding ACL2 model. First we define the corresponding function for the variable s: `(defun s (j) (bv-nat-be (b-and (nat-bv-be j) *mask*)))`

Actually, (s *j*) computes *(mod j 16)*. *bv-nat-be*, *nat-bv-be* are conversion functions according to the big endian representation, *b-and* is a macro computing the logic *and* operation between two bits, and between two bit-vectors of possibly distinct lengths.

The computation of a word W_s is defined by function *word-spec*:

```
(defun word-spec (j m-i)
;computes the W_(mod j 16) of the block m-i, for (≤ 16 j)
     (rotl 1 (b-xor
             (nth (bv-nat-be (b-and (nat-bv-be (+ 13 (s j))) *mask*)) m-i)
             (nth (bv-nat-be (b-and (nat-bv-be (+ 8 (s j))) *mask*)) m-i)
             (nth (bv-nat-be (b-and (nat-bv-be (+ 2 (s j))) *mask*)) m-i)
             (nth (bv-nat-be (b-and (nat-bv-be j) *mask*)) m-i))))
```

Temp-spec computes the variable TEMP for the step *j* of algorithm:

```
(defun temp-spec (j working-variables m-i)
      (plus (rotl 5 (nth 0 working-variables))
            (F j (nth 1 working-variables) (nth 2 working-variables)
                (nth 3 working-variables))
```

```
          (nth 4 working-variables) (nth (s j) m-i) (K j)))
```

Finally, *digest-one-block-spec*, computes the variables a, b, c, d, e, for j steps of the algorithm:

```
(defun digest-one-block-spec (j working-variables m-i)
     (declare (xargs :measure (acl2-count (- 80 j))))
     (if (natp j)
         (cond ((<= 80 j) working-variables)
            ;80 steps of the algorithm have been computed,
            (t (digest-one-block-spec (+ 1 j)
            ;one computation step
                (temp-spec j working-variables
                      (if (<= 16 j) (repl (s j) (word-spec j m-i) m-i) m-i))
                ;the computation of a with the new computed W(mod j 16)
                (nth 0 working-variables) (rotl 30 (nth 1 working-variables))
                (nth 2 working-variables) (nth 3 working-variables)
                ;the computation of b, c, d, e
                (if (<= 16 j) (repl (s j) (word-spec j m-i) m-i) m-i))))
                ;the word W(mod j 16) is replaced with the new computed one
  nil))
```

(**repl** *i el list*), replaces the i-th element of *list* with *el*.

When one block has been processed, the values of H_i are updated:

$H_0 = H_0$+A; $H_1 = H_1$+B; $H_2 = H_2$+C; $H_3 = H_3$+D; $H_4 = H_4$+E;

The function *digest-spec* computes the digest for a padded message m:

```
(defun digest-spec (m hash-values)
     (if (endp m) hash-values
         (digest-spec (cdr m)
               ;the digest computed for a block becomes
               ;the initial hash value for the next one
               (intermediate-hash hash-values
                     (digest-one-block-spec 0 hash-values
                             (parsing (car m) 32))))))
```

The function *intermediate-hash* updates the hash values H_i.

When the last block has been processed, the message digest contains the last values of H_0, H_1, H_2, H_3, H_4.

Sha_norm defines the specification for SHA-1: an arbitrary message m is first padded, then it is parsed into blocks of 512 bits. The list of blocks is digested with the algorithm constants *h0*, *h1*, *h2*, *h3*, *h4* as initial hash values.

```
(defun sha-norm (m)
     (digest-spec (parsing (padding-1-256 m) 512) (list *h0* *h1* *h2* *h3* *h4*)))
```

Validation of the formal functional specification

The SHA-1 ACL2 model is executed on the test benches given in the standard document to check that the returned result is as expected. A complementary validation is obtained by proving the mathematical properties of the algorithm, using the ACL2 theorem prover [2]. In fact, a more general model has been written, to capture the common principles of the four versions of the SHA

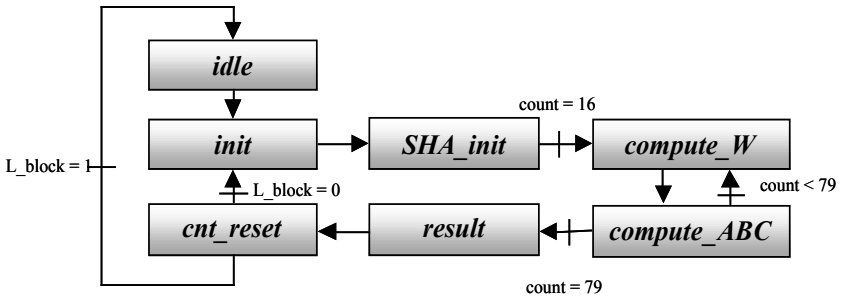

Figure 3: State machine

algorithm: SHA-1, SHA-256, SHA-384 and SHA-512, which differ essentially in the sizes of the message blocks, word, and digest. Seventy function definitions and over a hundred lemmas were written. Among the safety theorems that were proven for the SHA-1:

- The length of the padded message is a non-zero multiple of 512.
- The last 64 bits of the padded message represent the binary coding of the length.
- The first L bits of the padded message represent the initial message.
- The bits between the end-of-the-message bit and the last 64 bits are all 0.
- After parsing the padded message, the result is a vector of blocks, each of 512 bits.
- The final result of the SHA-1 is a five 32-bit words message digest.

Due to the nature of the digest computation, there is no straightforward algebraic expression for it; thus, the digest validation consisted in showing properties of the result of each processing step rather then proving its equivalence with a mathematical function.

3 SHA-1 Implementation

3.1 Main characteristics of the VHDL design

The VHDL model is written at the RTL level. The SHA core is composed of a control machine and a data path. The data path contains a compact description of the operators necessary for the digest computation. The transition graph of the control automaton for the state machine is shown on Figure 3.

The global behaviour is the following:

- *idle* is the wait state;
- *init* loads the constants H_0 to H_4 into the registers that store A, B, C, D, E;
- *SHA_init* computes the first sixteen values of A, B, C, D, E, i.e. the first sixteen steps of the algorithm;
- *compute_W* computes one of the 64 remaining words, W;
- *compute_ABC* computes the values of A, B, C, D, E corresponding to the previous W; the states *compute_W* and *compute_ABC* are repeated 64 times;
- *result* adds the last values of A, B, C, D, E to the last values of H_0 to H_4 respectively;

- *cnt_reset* resets the different counters; When the last block has been processed (L_block = 1) then signal *done* is set to 1, indicating that the message digest is available.

The design is fully synchronous. The pin description of the core is given in Table 1. The design also has 23 internal memories.

start, reset	input	bit	start signal, asynchronous core reset
reset_done	input	bit	invalidates the output *done*
clk	input	bit	clock signal
rdata	input	32-bit	Input data
base_addr	input	12-bit	first word W RAM address
nb_block	input	6-bit	number of blocks
addr	output	12-bit	RAM address
ram_sel, ram_write	output	bit	RAM select and write signals
wdata	output	32-bit	computed W to store into the RAM
busy	output	bit	core busy by digest computation
done	output	bit	digest message available
aout, bout, cout, dout, eout	output	32-bit	message digest output

Table 1: Pin description of the SHA core

3.2 Formal model of the SHA-1 design into ACL2

The VHDL is automatically translated into a functional model using a method based on symbolic simulation developed by our team [5]. The model is simulated symbolically for one clock cycle, actually corresponding to several VHDL simulation cycles, to extract the transition function for each output and state variable of the design. The body of a transition function is an if_expression, an arithmetic or a Boolean expression. The functions are translated into Lisp and used to define the Moore machine for the initial VHDL description. Here is the body of the corresponding transition function for the *done* signal.

```
(defun nextsig_done (reset reset_done start state cnt bl done)
   (if (equal reset 1) 0
       (if (equal state *idle*)
           (if (or (equal start 1) (equal reset_done 1))
               0 done)
        (if (and (equal state *resultw*)
                 (equal cnt 0)
                 (equal bl (list 0 0 0 0 0 0)))
             1 done))))
```

where, *idle*, *resultw* are user defined constants (in VHDL) to identify the states of the finite state machine of the VHDL design, *reset* and *reset_done* are inputs, *state*, *cnt* and *bl* are internal memories, and *done* is an output.

A state of the Moore machine is the set of all internal memories and all the outputs of SHA. A step is modeled as a function *Sim_step* which takes as parameters the inputs of SHA and the state of the machine at clock cycle k, and which produces the state of the machine at clock cycle k+1 (k is a natural). The body of *Sim_step* is the composition of the transition functions obtained by symbolic simulation.

We also extract from the VHDL the types for all input, internal and output objects. These informations are translated into ACL2 as two predicates *hyp_input* and *hyp_mem*. Here is the

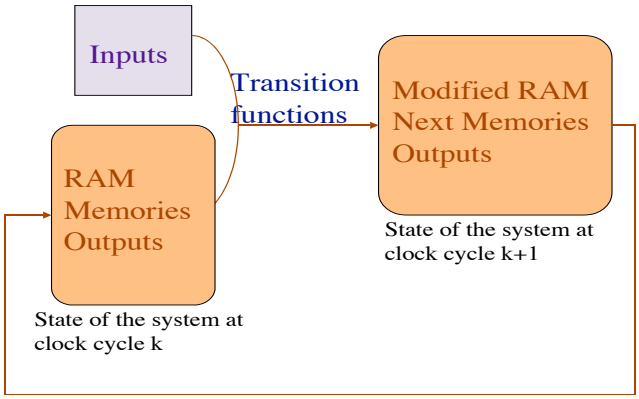

Figure 4: The Moore machine for SHA

corresponding *hyp_input* for the SHA-1 design.

```
(defun hyp-input (input)
   (and (bitp (nth *reset* input))
        (bitp (nth *reset_done* input))
        (bitp (nth *start* input))
        (wordp (nth *base_addr* input) 12)
        (wordp (nth *nb_bloc* input) 6)
        (wordp (nth *ram_rdata32* input) 32)))
```

where, *input* is a list of inputs, *reset*, *reset_done*, etc., are ACL2 defined constants that identify the position of signals *reset*, *reset_done*, etc., in the input list; (**bitp** *b*) is a predicate that states *b* is a bit, (**wordp** *bv n*) states *bv* is a bit vector of length *n*.

The SHA-1 design implements only the digest computation and it takes as input the message already padded from an external RAM. The RAM is also used to store the modified W_j computed during the digest.

The RAM (modeled as an alist) has the following behaviour:

- if the *ram_sel* bit is 1, the RAM is enabled, i.e. reading or writing is allowed;
- writing is allowed only if the *ram_write* bit is 1, and in this case the data *wdata* is written at the address *addr*;

```
(defun write-ram (mem ram)
   (if (and (equal (nth *ram_sel* mem) 1)
            (equal (nth *ram_write* mem) 1))
       (bind-equal (nth *addr* mem) (nth *wdata* mem) ram)
       ram))
```

- if the *ram_write* bit is 0, reading is allowed, and the data from the address *addr* is put in *rdata* which is one of the inputs for the design.

```
(defun read-ram (mem ram)
   (if (and (equal (nth *ram_sel* mem) 1)
            (equal (nth *ram_write* mem) 0))
```

```
        (binding-equal (nth *addr* mem) ram)
        nil))
```

The RAM is added to the state of the Moore machine (Figure 4).

The circuit is defined by the function *sha_vhdl* which has two parameters: the sequence of inputs *L_input* and the state *st*. *st* is composed of two parts: *memory* is the set of all internal and output signals of the SHA-1 and *ram* models the external RAM. The length of *L_input* gives the number of clock cycles, and *L_input* represents the list of symbolic or numeric values for the SHA-1 input ports at each clock cycle:

(**list** inputs_cycle-1 inputs_cycle-2 ... inputs_cycle-k)

If the inputs list is empty, the computation is finished and *sha_vhdl* returns the state *st*. Otherwise, the next state is computed by calling the step function *Sim_step*, then *memory* and *ram* are updated. Again, this model is executable, and we have initially checked it using the test benches provided in the SHA standard.

```
(defun sha_vhdl (L_input st)
    (if (atom L_input) st
        (let* ((memory (car st))
               (ram (cdr st))
               (new-mem
                   (Sim_step (cons (read-ram memory ram) (car L_input))
                             memory)))
              (sha_vhdl (cdr L_input)
                   (cons new-mem (write-ram new-mem ram))))))
```

4 Proof of correctness of the implementation

At this point, we have two models of SHA-1 in ACL2: the translation by hand of the standard FIPS-180-2 *sha_norm*, and the automatic translation of the VHDL description *sha_vhdl*.

To prove that the VHDL implementation is compliant with the functional specification, we must show that for any arbitrary input message the execution of *sha_vhdl* for the appropriate input and the appropriate number of clock cycles (until the computation is done) returns the same message digest as the one returned by *sha_norm*.

First we define the environment in which *sha_vhdl* is executed, then we compute the result of the execution of *sha_vhdl* until the final state and we prove it is equivalent with some intermediate digest functions, and at last we prove the relation between the results computed by the two models (Theorem 10 below).

One difficulty in performing the proof is brought by the presence of the RAM in the implementation and by the added circuitry to access the informations and write back partial results during the computation. Moreover, a 512-bit block is overwritten by the W_i ($16 \leq i \leq 79$) words during the digest computation (see Figure 2). In contrast, the specification is a functional algorithm that processes each message block without side-effect.

Significant human time was spent identifying the set of intermediate functions and their properties, the composition of which could be proven equivalent to the specification on the one side, each one corresponding to an abstraction of the behavior of the VHDL state machine, starting from some state and for a given number of cycles.

The inputs considered for *sha_vhdl* are described in Table 2, where X stands for "don't care", *nb_bloc* is the 6-bit representation of the natural number *nb*: the number of blocks to be processed, *base_addr* is a bit-vector of size 12. *nb_bloc* and *base_addr* are symbolic.

We consider L_input as (**list*** *input_cycle_1 input_cycle_2 input*).

Cycle	**1**	**2**	**3**	...
Input	*input_cycle_1*	*input_cycle_2*	input	
Reset	1	0	0	...
Start	X	1	X	...
Reset_done	X	X	X	...
Nb_bloc	X	nb_bloc	nb_bloc	...
Base_addr	X	X	base_addr	...

Table 2: The symbolic input for *sha_vhdl*

The RAM is modeled by the predicate *wordp-ram*:

```
(defun wordp-ramp (ram n)    ;recognizes an alist of n-bit words
   (if (alistp ram)
       (if (endp ram) nil
          (if (and (consp ram) (endp (cdr ram)))
              (wordp (cdar ram) n)
           (and (wordp (cdar ram) n)
                (wordp-ramp (cdr ram) n))))
     nil))
```

In our case, (**wordp-ramp** *ram* 32).

The RAM zone that starts at address *base_addr* is of particular interest, as it contains the message to be processed by the SHA-1 design. We consider the RAM as being the concatenation of two RAMs: (**append** *unknown_ram base_addr_ram*), where *unknown_ram* is an alist of no interest for the computation and *base_addr_ram* is the zone starting from *base_addr*. The domains of the two RAMs are disjoint. We define a predicate *ramp-base* that recognizes *base_addr_ram*.

```
(defun ramp-base (base_addr base_addr_ram)
    (declare (xargs :measure (len ram)))
   (if (alistp base_addr_ram)
       (if (endp base_addr_ram) nil
           (if (and (consp base_addr_ram)
                    (endp (cdr base_addr_ram)))
               (equal (caar base_addr_ram) base_addr)
             (and (equal (caar base_addr_ram) base_addr)
                  (ramp-base (plus base_addr 1)
                             (cdr base_addr_ram)))))
       nil))
```

where *plus* is the + operation, overloaded with the addition between two unsigned bit-vectors, and between an unsigned bit-vector and a natural number.

Sha-vhdl needs 3 clock cycles to initialize the system and set A, B, C, D, E to their initial values; then it needs 342 clock cycles to compute the digest for one block. The 342 cycles are decomposed as: 16 for reading the first 16 words and computing 16 steps of digest, 320 to compute an intermediate digest, 3 to combine the results with the initial hash values of the block, 2 to store the message digest obtained so far. The last cycle returns to the digest computation for the next block, or to the *idle* state. So, in order to process *nb* blocks, the design needs $3 + (342 * nb)$ clock cycles.

The above could let the reader believe that we are performing simulation. This is not the case. A stepwise approach is developed, which proves intermediate theorems for each main computation step of the overall *sha_vhdl*. A computation step corresponds to a state of the VHDL state machine (Figure 3).

Each computation step is first generalized to perform some function for an arbitrary number k of clock cycles and then proved by induction on k and on the circuit state. Specific induction schemes had to be constucted for this purpose. Then k is instanciated to the actual number of cycles performed by the circuit (16 for SHA_init, 320 for looping between compute_w and compute_ABC, etc... see Figure 3). The reasoning engine considers the initial value of all memories and registers as arbitrary, and *nb* (the number of blocks) to be an unbounded (but finite) natural integer.

The overall proof is decomposed as a set of theorems that prove the result of the VHDL design, for a number of clock cycles, equal with some intermediate functions:

- Theorems 1 to 3 establish the results of the initialization phases;
- Theorem 4 corresponds to the first 16 computation steps: the RAM is unchanged;
- Theorem 5 corresponds to the subsequent 64 steps: the block is overwritten in the RAM;
- Theorems 6 and 7 update the block digest and initialize the computation for the next block;
- Theorem 8 combines Theorems 3 to 7 to establish the result of 342 clock cycles over one block;
- Theorem 9 combines Theorems 1, 2, 8 to establish the result of the VHDL computation for nb blocks (over 3+342*nb cycles);
- finally, Theorem 10 says that the result considered in Theorem 9 is equal to the message digest computed by the specification.

Because of the big number of state variables, we only refer to some of them, which seem important to illustrate our approach: *a*, *b*, *c*, *d*, *e* are 32-bit registers storing the digest computation, *state* is a 3-bit vector giving the state of the VHDL finite state machine, *bl* is a 6-bit word representing the number of blocks that remain to be processed, *count* is a 8-bit word counting the iterations of the digest. In the following theorems, all these variables are components of the global state *st*.

The following theorems hold for the input and the RAM defined as above.

Theorem 1 *Starting from an arbitrary state, after one cycle, with input_cycle_1 as input, the system is in the idle state and the RAM is not modified.*

Proof: By symbolic execution of *sha_vhdl* for one clock cycle:
`(sha_vhdl (list (car L_input)) st)`, where *st* is an arbitrary state. □

Theorem 2 *Starting from state idle, after 2 cycles, the system is in state init, the state variables a, b ,c, d, e are initialized with the initial hash values: h0, h1, h2, h3, h4, which are constants of the standard, bl is initialized with the number of block to be processed, i.e. nb_bloc, all the other state variables are initialized and the RAM is not modified.*

Proof: First we execute symbolically *sha_vhdl*, with *input_cycle_2* as input, for one clock cycle. Then, starting from the resulting state, we execute *sha_vhdl* once more for one clock cycle, providing (car input) as input. We combine the two theorems using the property:

`(equal (sha_vhdl (firstn i in) (sha_vhdl (firstn j in) st))`
`(sha_vhdl (firstn (+ i j) in) st))`, *i*, *j* naturals and *(≤ (+ i j) (len in))*.

□

Theorem 3 *Starting from state init, after 1 cycle the system is in the SHA_init state (i.e. the computation can begin), count="00000000" and the memory has not changed.*

Proof: By symbolic execution of *sha_vhdl* for one clock cycle. □

Theorem 4 *Starting from the initial computation state (state=SHA_init, count="00000000"), after 16 cycles, the RAM is not modified and the system variables verify the following: state=compute_W, count="00010000" and a, b, c, d, e, hold the result of the first 16 steps of the digest computation:*

```
(digest-one-block-impl 16 '(0 0 0 0 0 0 0 0)
      (nth *a* (memory st)) (nth *b* (memory st))
      (nth *c* (memory st)) (nth *d* (memory st))
      (nth *e* (memory st)) (ram st)
      (nth *base_addr* (car input)) (nth *nb_bloc* (car input))
      (nth *bl* (memory st)))
```

where (memory st) is (car st), (ram st) is (cdr st).

Proof: See Appendix.

Theorem 5 *Starting from the computation state for the first word (state=compute_W, count= "00010000"), after 320 cycles, the RAM is modified as described by function modified-ram, and the system variables verify the following: state=result, count="01010000", and a, b, c, d, e hold the result of the last 64 steps of the digest computation:*

```
(digest-one-block-impl 64 '(0 0 0 1 0 0 0 0)
      (nth *a* (memory st)) (nth *b* (memory st)) (nth *c* (memory st))
      (nth *d* (memory st)) (nth *e* (memory st)) (ram st)
      (nth *base_addr* (car input)) (nth *nb_bloc* (car input))
      (nth *bl* (memory st)))
```

```
(defun modified-ram (i count ram base_addr nb_bloc bl)
   (if (zp i) ram
       (if (<= 16 (bv-nat-be count))
           (modified-ram (- i 1) (plus 1 count)
                 (bind-equal (next-addr-word nb_bloc bl base_addr count 0)
                 ;the new computed word is written in the RAM at its corresponding address
                     (word-impl count base nb_bloc bl ram)
                     ram)
                 base nb_bloc bl)
         (modified-ram (- i 1) (plus 1 count) ram base nb_bloc bl))))
```

Proof: See Appendix.

Theorem 6 *Starting from the result state (state=result), after 3 cycles, the system variables verify the following: state=cnt_reset, the number of blocks to be processed, bl, is decremented and a, b, c, d, e are added to h0, h1, h2, h3, h4, which are intended to hold the hash values during the computation.*

Proof: By symbolic execution of *sha_vhdl* for one clock cycle. □

Theorem 7 *Starting from the reseting state (state=cnt_reset), after 2 cycles, if the number of blocks to be processed is higher than 0, then state=init and count is reset to "00000000", otherwise state=idle , done is 1 and the values of a, b, c, d, e are available as output.*

Proof: By symbolic execution of *sha_vhdl* for one clock cycle. □

By combining Theorem 3 to 7, we have:

Theorem 8 *Starting from the initial state (state=init), after 1+16+320+3+2=342 clock cycles, if the number of blocks to be processed is higher than 0, then the system is in the initial state (state=init), otherwise it is in the idle state (state=idle), and in both cases RAM is*

```
(modified-ram 80 '(0 0 0 0 0 0 0 0) (ram st)
    (nth *base_addr* (car input)) (nth *nb_bloc* (car input)))
```

and a, b, c, d, e hold the result of the digest for one block:

```
(intermediate-digest
   (list (nth *a* (memory st)) (nth *b* (memory st))
            (nth *c* (memory st)) (nth *d* (memory st))
            (nth *e* (memory st)))
   (digest-one-block-impl 80 '(0 0 0 0 0 0 0 0)
            (nth *a* (memory st)) (nth *b* (memory st))
            (nth *c* (memory st)) (nth *d* (memory st))
            (nth *e* (memory st)) (ram st)
            (nth *base_addr* (car input)) (nth *nb_bloc* (car input))
            (nth *bl* (memory st))))
```

Now we can prove the general computation theorem for *sha_vhdl*:
Let n be the number of blocks of the message, nb=(bv-nat-be (nth *nb_bloc* (car input))).

Theorem 9 *Starting from any state, for any message of n blocks, stored in RAM at address base_addr, after the execution of sha_vhdl for (+ 3 (* 342 nb)) clock cycles, the system is in its final state (done = 1) and the values of the output are equal to the result of digest-impl on the same message:*

```
(digest-impl (list *h0* *h1* *h2* *h3* *h4*)
        (ram st) (nth *base_addr* (car input))
        (nth *nb_bloc* (car input)) (nth *nb_bloc* (car input)))
```

Proof: See Appendix.

Now let us prove that the implementation is compliant with the specification model.
A little problem arises because *sha_vhdl* uses as input the message already padded, contrary to *sha_norm* which has as input the initial message. So, instead of comparing *sha_vhdl* with *sha_norm*, we compare it with *digest-spec*.

Theorem 10 *Starting from any state, for any message of nb blocks stored in RAM at address base_addr, after the execution of sha_vhdl for (+ 3 (* 342 nb)) clock cycles, the system is in its final state (done = 1) and the values of the output are equal to the result of the digest-spec applied to the message parsed in blocks of 512 bits:*

```
(digest-spec
    (parsing (concat (get-i-ram (* 16 (bv-nat-be (nth *nb_bloc* (car input))))
                                (nth *base_addr* (car input)) (ram st))) 512)
    (list *h0* *h1* *h2* *h3* *h4*))
```

Proof: As the padded message to be digested, M, is stored into the RAM,
$M =$ `(concat (get-i-ram (* 16 nb_bloc) base_addr ram))`,
where (**get-i-ram** *i addr ram*) returns the list of i words from *ram*, starting from address *addr*, and (**concat** l) concatenates all elements of l.

```
(defun get-i-ram (i addr ram)
   (if (zp i) nil
       (cons (binding-equal addr ram)
             (get-i-ram (- i 1) (plus 1 addr) ram))))
```

The block i of M, M_i is (get-i-ram 16 (plus base_addr (* 16 i)) ram). The current block i is *(bv-nat-be (minus nb_bloc bl))*.

Using Theorem 9, this comes to prove:

```
(equal (digest-impl (list *h0* *h1* *h2* *h3* *h4*)
                    (ram st) (nth *base_addr* (car input))
                    (nth *nb_bloc* (car input)) (nth *nb_bloc* (car input)))
       (digest-spec
               (parsing (concat
                   (get-i-ram (* 16 (bv-nat-be (nth *nb_bloc* (car input))))
                              (nth *base_addr* (car input)) (ram st))) 512)
               (list *h0* *h1* *h2* *h3* *h4*)))
```

We first prove a generalized form of the property above: the implementation model and the specification model computes the same message digest for $k =$ *(bv-nat-be (minus nb_bloc bl))* blocks of message, and for same initial hash values.

```
(equal (digest-impl hash-values ram base_addr nb_bloc bl)
       (digest-spec
          (parsing (concat
             (get-i-ram (* 16 (bv-nat-be bl))
                        (plus base_addr (* 16 (bv-nat-be (minus nb_bloc bl))))
                        ram)) 512)
           hash-values)
```

This is proven using the induction scheme generated by *digest-impl*.

A large number of lemmas are needed in order to complete the proof, only the top level ones are briefly described:

Lemma 1 *The result of the computation of the digest of one block is the same in both specification and the implementation model:*

```
(equal (digest-one-block-impl 80 '(0 0 0 0 0 0 0 0) a b c d e ram base_addr nb_bloc bl)
       (digest-one-block-spec 0 (list a b c d e)
            (get-i-ram 16 (plus base_addr
                        (* 16 (bv-nat-be (minus nb_bloc bl)))) ram)))
```

Proof: We first prove a generalized form of the lemma:

```
(equal (digest-one-block-impl (- 80 (bv-nat-be count))
            count a b c d e ram base_addr nb_bloc bl)
       (digest-one-block-spec (bv-nat-be count) (list a b c d e)
             (get-i-ram 16 (plus base_addr
                                (* 16 (bv-nat-be (minus nb_bloc bl)))) ram)))
```

This is proven using the induction scheme generated by *digest-one-block-impl*.
For the proof to succeed, several properties are needed:

- The computation of the word $W_{(mod\ count\ 16)}$ is the same in both the specification and the implementation model:

```
(equal (word-impl count base_addr nb_bloc bl ram)
       (word-spec (bv-nat-be count)
                  (get-i-ram 16
                             (plus base_addr(* 16 (bv-nat-be (minus nb_bloc bl))))
                             ram))
```

- The left rotating operations in both implementation and specification model are equal:

```
(equal (append (segment 5 32 a) (segment 0 5 a))
            (rotl 5 a))
```

```
(equal (next-b b) (rotl 30 b))
```

- The logical function used in the implementation is equal to the specification defined one:

```
(equal (f-impl count b c d) (f (bv-nat-be count) b c d))
```

- In both models the replacement of the word $W_{(mod\ count\ 16)}$ with the new computed one has the same effect on the initial message:

```
(equal (get-i-ram 16 (plus base_addr (* 16 (bv-nat-be (minus nb_bloc bl))))
            (bind-equal (next-addr-word nb_bloc bl base_addr count 0)
                        (word-impl count base_addr nb_bloc bl ram) ram))
       (repl (s (bv-nat-be count))
             (word-spec (bv-nat-be count)
                  (get-i-ram 16 (plus base_addr
                                      (* 16 (bv-nat-be (minus nb_bloc bl))))
                             ram))
             (get-i-ram 16 (plus base_addr
                                 (* 16 (bv-nat-be (minus nb_bloc bl))))
                        ram)))
```

This is an instantiation of a more general property:

```
(equal (get-i-ram n addr (bind-equal (plus addr x) y ram))
       (repl (bv-nat-be x) y (get-i-ram n addr ram)))
```

□

Lemma 2 *After processing k blocks, the memory storing the rest of the message to be processed (starting from the address (plus base (* 16 k))) is unaltered.*

```
(equal (get-i-ram n (plus base i (* 16 (bv-nat-be (minus nb_bloc bl))))
           (modified-ram j count ram base nb_bloc bl))
       (get-i-ram n (plus base i (* 16 (bv-nat-be (minus nb_bloc bl))))))
```

Proof: By induction using the scheme:

```
(defun induction-scheme-get-i-ram (n i)
   (if (or (zp n) (zp i)) t
     (induction-scheme-get-i-ram (1- n) (1+ i))))
```

□

All theorems use a large number of properties that we proved about bit-vectors and operations with bit vectors (logical, arithmetic, concatenation, shifting, conversions, etc), about the RAM and L_input.

5 Conclusion

The benefits of our research for the project are multiple. For proving a SHA-1 circuit, we produced executable and reusable specifications for a standard algorithm that was previously available under an informal notation only and we also developed a stepwise method for the proof of the implementation vs specification, based on the state machine of the RTL. This approach can be reused for other control - data path combined designs. The proof of correctness of the RTL implementation with respect to the functional specification for an arbitrary message, could be performed only using theorem proving techniques, and involved significant human time. The investment was however justified by the high degree of security expected from the circuit application. A library of functions and theorems over bit vectors was developed: this library is reusable for other applications.

The concurrent development of the design and of the proof were beneficial: the modeling effort was performed in close interaction with the design, and could provide early feed-back on the design style.

A couple of errors were also uncovered in the initial VHDL, the most serious being an excessive number of cycles in the digest computation. The use of an executable logic was key to the successful validation of both the specification and the RTL, as it provides an easy model debugging facility.

We believe that the design of a reusable core module should increasingly come with its formal proof of correctness: our work demonstrates that this is feasible, and shows a strategy to reach this goal.

References

[1] National Institute of Standards and Technology: "Secure Hash Standard", Federal Information Processing Standards Publication 180-2, 2002.

[2] D. Toma, D. Borrione: "SHA Formalization", ACL2 Workshop, Boulder, USA, 13-14 July 2003.

[3] C. Chavet: Modélisation et validation d'une architecture embarquée sur la puce pour les applications sécuritaires, Msc Thesis,Université Joseph Fourier, June 2003.

[4] M. Kaufmann, P. Manolios, and J S. Moore: Computer-Aided reasoning: ACL2 An approach.(Vol.1) and ACL2 case Studies (Vol.2), Kluwer Academic Press, 2000.

[5] D. Toma, D. Borrione, G. Al-Sammane: Combining several paradigms for circuit validation and verification, (to appear), CASSIS 2004. accessible also as a research report from http://tima.imag.fr/publications.

Appendix: Proof of theorems

A.1 Proof of theorem 4

We define the implementation digest for one block as:

```
(defun digest-one-block-impl (i count a b c d e ram base_addr nb_bloc bl)
  (if (zp i) (list a b c d e)
      (digest-one-block-impl
         (- i 1) (plus count 1)
         (temp-impl count a b c d e
             (if (<= 16 (bv-nat-be count))
                 (word-impl count base_addr nb_bloc bl ram)
               (binding-equal (next-addr-word nb_bloc bl base_addr count 0) ram)))
         a (next-b b) c d
         (if (<= 16 (bv-nat-be count))
             (bind-equal (next-addr-word nb_bloc bl base_addr count 0)
                         (word-impl count base_addr nb_bloc bl ram) ram)
              ram)
          base_addr nb_bloc bl)))
```

Where:

- *temp-impl* computes the intermediate variable TEMP,

```
(defun temp-impl (count a b c d e data)
  (plus (F-impl count b c d) e data
         (append (segment 5 32 a) (segment 0 5 a))
         (K (bv-nat-be count))))
```

- *next-addr-word* computes the address of $W_{(mod(plus\ i\ count)16)}$ relative to *base_addr*. The words of the block k=(bv-nat-be (minus nb_bloc bl)) are read in these 16 cycles. *minus* is the - operation overloaded with the substraction between two bit-vectors, and between a bit-vector and a natural.

```
(defun next-addr-word (nb_bloc bl base_addr count i)
   (plus (plus base_addr (plus i (segment 4 8 count)))
         (* 16 (bv-nat-be (minus nb_bloc bl )))))
```

- *word-impl* computes $W_{(mod\ count\ 16)}$, if ($\leq$ 16 count):

```
(defun word-impl (count base_addr nb_bloc bl ram)
  (shift (b-xor
    (binding-equal (next-addr-word nb_bloc bl base_addr count 0) ram)
    (binding-equal (next-addr-word nb_bloc bl base_addr count 2) ram)
    (binding-equal (next-addr-word nb_bloc bl base_addr count 8) ram)
```

```
(binding-equal (next-addr-word nb_bloc bl base_addr count 13) ram)))
```

To prove Theorem 4 we first prove a more general form:

Theorem 4 generalized *Starting from any of the first 16 computation states* (*state=SHA_init*, `(≤ (+ j count) 16)`, `(≤ 1 j)`), *after j clock cycles*,
if `(equal (+ j count) 16)` *then state=compute_W, otherwise state=SHA_init.*
In both cases a,b,c,d,e hold the result of j digest steps computed by the function digest-one-block-impl, count is incremented by j and the RAM is unchanged.

The theorem is proved by induction using the scheme:

```
(defun induction-scheme-1-cycle (j input st)
    (if (or (zp j) (endp input)(atom st)) t
      (induction-scheme-1-cycle (1- j)
                (nthcdr 1 (firstn j input)) (sha_vhdl (firstn 1 input) st))))
```

The following property of *digest-one-block-impl* is needed for the proof to succeed:
If *(and (≤ (+ i j count) 16) (≤ 1 i))* , *i*, *j* naturals, let *digest-j-steps* be equal to *(digest-one-block-impl j count a b c d e ram base_addr nb_bloc bl)*, then

```
(equal (digest-one-block-impl i (plus count j)
           (car digest-j-steps) (nth 1 digest-j-steps) (nth 2 digest-j-steps)
           (nth 3 digest-j-steps) (nth 4 digest-j-steps)
           ram base_addr nb_bloc bl)
      (digest-one-block-impl (+ i j) count a b c d e ram base_addr nb_bloc bl))
```

The proof also uses several lemmas on input, ram and bit-vector operations.

In the generalized theorem, if *j* is instantiated with 16 and *count* with "00000000", we obtain the theorem to prove.□

A.2 Proof of theorem 5

The 5*64=320 cycles are needed to compute the digest of a block: 5 cycles for the computation of TEMP (the intermediate variable) and the algorithm must be applied 64 times to compute the intermediate digest.

First we prove a generalized version of the theorem:

Theorem 5 generalized *Starting from any word computation state* (*state=compute_W*), `(≤ (+ j count) 80)` and `(≤ 1 j)`), *after (* 5 j) cycles, if (equal (+ j count) 80) then state=result, otherwise state=compute_W.*
In both cases a,b,c,d,e hold the result of j digest steps computed by the function digest-one-block-impl, count is incremented by j, and RAM is
`(modified-ram j count (ram st) (nth *base_addr* (car input)) (nth *nb_bloc* (car input))).`

The theorem is proved by induction using the scheme:

```
(defun induction-scheme-5-cycles (j input st)
 (if (or (zp j) (endp input)(atom st)) t
   (induction-scheme-5-cycles (1- j)
              (nthcdr 5 (firstn (* 5 j) input))
```

```
(sha_vhdl (firstn 5 input) st))))
```

Some intermediate theorems are needed to compute the behaviour of *sha_vhdl* for 5 cycles, starting from *compute_W* state:

Theorem 5.1 *Starting from any word computation state (state=compute_W), after 4 cycles, the corresponding* $W_{(mod\ count\ 16)}$ *word is computed by word-impl, state=compute_ABC, count and RAM are unchanged.*

Theorem 5.2 *Starting from any variables computation state (state=compute_ABC, (≤ count 79)), after one cycle*

- if count is 79 then the algorithm finished and state=result,

- otherwise, the algorithm was applied less than 79 times and state=compute_W;

In both cases count is incremented, TEMP is computed and $W_{(mod\ count\ 16)}$*:*

```
(word-impl (nth *count* (memory st)) (nth *base_addr* (car input))
           (nth *nb_bloc* (car input)) (nth *bl* (memory st)) (ram st))
```

is written in RAM at the address

```
(next-addr-word (nth *nb_bloc* (car input)) (nth *bl* (memory st))
                (nth *base_addr* (car input)) (nth *count* (memory st)) 0).
```

The following properties of *digest-one-block-impl* and *modified-ram* are also needed for the proof to succeed:

For i, j naturals:

- Let *digest-j-steps* be equal to *(digest-one-block-impl j count a b c d e ram base_addr nb_bloc bl)*, then:

```
(equal (digest-one-block-impl i (plus count j)
                 (car digest-j-steps) (nth 1 digest-j-steps) (nth 2 digest-j-steps)
                 (nth 3 digest-j-steps) (nth 4 digest-j-steps)
                 (modified-ram j count ram base nb_bloc bl)
                 base_addr nb_bloc bl)
          (digest-one-block-impl (+ i j) count a b c d e ram base_addr nb_bloc bl))
```

- ```
  (equal (modified-ram i (plus count j)
                  (modified-ram j count ram base nb_bloc bl) base nb_bloc bl)
           (modified-ram (+ i j) count ram base nb_bloc bl))
  ```

In the generalized theorem, if j is instantiated with 64 and *count* with "00010000", we obtain the theorem to prove. □

A.3 Proof of theorem 9

We define the general implementation digest function as:

```
(defun digest-impl (hash-values ram base_addr nb_bloc bl)
   (if (equal (bv-nat-be bl) 0) hash-values
     (digest-impl
          (intermediate-digest hash-values
             (digest-one-block-impl 80 '(0 0 0 0 0 0 0 0)
                       (car hash-values) (nth 1 hash-values) (nth 2 hash-values)
                       (nth 3 hash-values) (nth 4 hash-values)
                       ram base_addr nb_bloc bl))
```
```

```
(modified-ram 80 `(0 0 0 0 0 0 0 0) ram base_addr nb_bloc bl)
base_addr nb_bloc (minus bl 1))))
```

We prove the Theorem 9 using Theorem 1, Theorem 2 and by instantiating *bl* with *nb_bloc* in a more general form of the theorem:

Theorem 9 generalized *Starting from the initial state (state=init), with (≤ bl nb_bloc) and (≤ 1 bl), after the execution of sha_vhdl for (* 342 bl) clock cycles, if bl and nb_bloc hold the same values, then the system is in its final state (done = 1), otherwise it is in the initial state. In both cases the RAM is modified, bl is decremented, and the values of a, b, c, d, e are equal to the result of digest-impl on the message stored in RAM starting from the address (plus base_addr (16 (minus nb_bloc bl))).*

The theorem above is proved by induction using the scheme:

```
(defun induction-scheme-digest (j input st)
    (if (or (zp k) (endp input)(atom st)) t
      (induction-scheme-digest (1- j)
                 (nthcdr 342 (firstn (* 342 j) input))
                 (sha_vhdl (firstn 342 input) st))))
```

where *j* is *(bv-nat-be (nth *bl* (memory st)))*.

The following property of *digest-impl* is needed for the proof to succeed:

If *(≤ (plus i j) nb_bloc)*, *i*, *j* 6-bit vectors

```
(equal (digest-impl
         (digest-impl hash-values ram base_addr nb_bloc i)
         (modified-ram 80 '(0 0 0 0 0 0 0 0) ram base_addr nb_bloc)
         base_addr nb_bloc j)
       (digest-impl hash-values ram base_addr nb_bloc (plus i j)))
```

□